Theoretical Computer Science 45 (1986) 63-86
North-Holland

TRANSDUCERS AND REPETITIONS

Maxime CROCHEMORE

Centre Scientifique et Polytechnique, Université de Paris-Nord, 93430 Villetaneuse, France

Communicated by M. Nivat
Received January 1986

Abstract. The factor transducer of a word associates to each of its factors (or subwords) their first occurrence. Optimal bounds on the size of minimal factor transducers together with an algorithm for building them are given. Analogue results and a simple algorithm are given for the case of subsequential suffix transducers. Algorithms are applied to repetition searching in words.

Résumé. Le transducteur des facteurs d'un mot associe à chacun de ses facteurs leur première occurrence. On donne des bornes optimales sur la taille du transducteur minimal d'un mot ainsi qu'un algorithme pour sa construction. On donne des résultats analogues et un algorithme simple dans le cas du transducteur sous-séquentiel des suffixes d'un mot. On donne une application à la détection de répétitions dans les mots.

Contents

Introduction

The general string-matching problem, searching for a pattern in a string, can be divided into two parts according to whether it is the pattern or the string which changes most often. When the pattern is fixed, efficiency is reached by preparing it. Knuth, Morris and Pratt's algorithm [11] together with Boyer and Moore's algorithm [8] are the best known examples for such a preparation. When the string is a dictionary, for instance, preprocessing it can highly improve further word searches. This approach has been initiated by Weiner [19], and McCreight [15]

gave a good practical algorithm for building the so-called suffix-tree of a string (the branches of the tree are labelled by the suffixes of the string).

Automata theory unifies the two approaches and yields an algorithm which is, in our opinion, both simple and efficient for constructing the right structure underlying Weiner's method.

Consider a word u which is to be searched for. A direct solution to string-matching is to build a deterministic automaton to recognize the set of words ending with u, say A^*u. Then we can make the automaton work on the string inside which u is expected, and get a real-time search algorithm [1, 2]. In this situation, time and space depend on the size of the alphabet A. Knuth, Morris and Pratt's algorithm does not explicitly build the above automaton but a representation of it by default states (see, e.g., [3]) of the minimal deterministic automaton for A^*u. It must be noted that the difference between their algorithm and the original Morris and Pratt algorithm [16] lies precisely in the choice of default states. Time and space become independent of the size of the alphabet, but the search still remains linear in the length of the pattern and the string.

Suffix-trees or other compacted trees have a size which is linear in the length of the string. Their construction [15, 19] depends inherently on the size of the alphabet. Suffix-trees have found a great number of applications (see, e.g., [4, 9, 17]), and Boyer and Moore's algorithm itself uses functions precomputed on the pattern and which are included in the suffix-tree of the mirror image of the pattern. This proves how powerful Weiner's method is for algorithmic problems concerning words.

In the same way, a direct solution to the problem is to build a deterministic automaton that recognizes all the factors (also called subwords) of the string to be searched. This would have been unrealistic and impractical without the surprising result of Blumer et al.: there exists an automaton which recognizes the factors of a word x and whose size is linear in the length of x and independent of the size of the alphabet. They gave, at the same time, a linear algorithm for building the automaton [7]. Starting from ideas included in Knuth, Morris and Pratt's paper [11], one can build the smallest automaton that recognizes the same language [10].

In most applications, however (including applications of suffix-trees), more information is needed from the automaton. So, it is quite natural to consider transducers that output positions of the factors inside the string. It turns out that the underlying automata of several interesting transducers of that kind are minimal automata. Their size and construction are linear in the length of the word considered.

For the transducers as well as for the automata in this paper we consider it to be important to distinguish whether they deal with factors or suffixes of the string. In general, the size of the minimal automaton is not the same in both cases. This also avoids considering a marker at the ends of words.

This paper is divided into eight sections. The first two sections are mainly devoted to notations and definitions which follow [5, 12]. In Section 3, the notion of suffix link is introduced, which is one of the key-points of the construction of factor transducers. First, bounds on the size of minimal factor transducers are established

with the help of suffix links. These bounds are improved up to optimality in Section 4 where the main Theorem 4.1 is proved. The proof of Theorem 4.1 guides the design of the algorithm that builds minimal factor transducers, in Section 5. Time complexity of the construction is discussed in Section 6. Then, the case of suffixes is examined in Section 7. Bounds and an algorithm, which is rather different and simpler than the first one in Section 5, are given for suffix transducers. The last section deals with a surprising application to repetitions in words which has to do with data compression methods such as that of Ziv and Lempel [20]: squarefreeness of words can be tested in linear time on a given alphabet. This result first proved in [9] with suffix-trees has been rediscovered by Main and Lorentz [14] using another method. Additional results needed by the application are given in Appendix A.

Many results of Sections 4 to 6 have been independently discovered by Blumer et al. and may be found in [6]. This paper also contains an interesting discussion on different algorithms following Weiner's approach.

1. Factor automata

All the words considered in this paper are elements of the free monoid A^* generated by some finite alphabet A. The empty word of A^* is denoted by 1 and A^+ is $A^* - \{1\}$. In the following, letters of A will be denoted by $a, b, c, \ldots$ and words of A^* by $x, y, z, u, v, w, \ldots$ We also use the notation $|x|$ for the length of a word x as well as for the cardinality of a set.

The set of factors (sometimes called subwords) of a given word x is

$$F(x) = \{y \in A^* \mid \exists u, v \in A^*, x = uyv\}.$$

The set of suffixes (also called right factors) of x is

$$S(x) = \{y \in A^* \mid \exists u \in A^*, x = uy\}.$$

An occurrence of a factor y of x is a position of y inside x and is formally defined as a triple (u, y, v) when $x = uyv$.

If $x = yz$, another useful notation is $y^{-1}x$ which denotes z, and xz^{-1} for y. If X is a set of words, quotients of X by a word x are

$$x^{-1}X = \{z \in A^* \mid xz \in X\} \quad \text{and} \quad Xx^{-1} = \{y \in A^* \mid yx \in X\}.$$

Sets $F(x)$ and $S(x)$, being finite, are recognized by minimal deterministic automata respectively denoted $\mathcal{F}(x)$ and $\mathcal{S}(x)$. We do not consider complete automata and minimal, for a deterministic automata, means to have the minimal number of states inside the class of all deterministic automata that recognize the same set of words. By doing so, the automata considered will also have the minimal number of edges or transitions labelled by letters of the alphabet A.

The formal definition of $\mathcal{F}(x)$ is a straightforward application of Myhill and Nerode's theorem on recognizable languages. Let R_x, or simply R if no confusion

can arise, be the equivalence relation on A^* defined by

$$yR_xz \text{ iff } \forall w \in A^*, yw \in F(x) \leftrightarrow zw \in F(x).$$

Equivalently, one has

$$yR_xz \text{ iff } y^{-1}F(x) = z^{-1}F(x).$$

Relation R_x is right invariant with respect to concatenation of words.

Let $R_x(y)$ (or $R(x)$) denote the equivalence class of word y. Then, states of $\mathcal{F}(x)$ are the equivalence classes of factors of x:

$$\{R(y) \mid y \in F(x)\},$$

the initial state is $R(1)$ and all states are terminal. We denote the transition function of $\mathcal{F}(x)$ by a dot and its definition is

$$\text{if } a \in A \text{ and } ya \in F(x),\ R(y).a = R(ya).$$

Right invariance of R makes this definition coherent. Figure 1 shows an example of factor automaton.

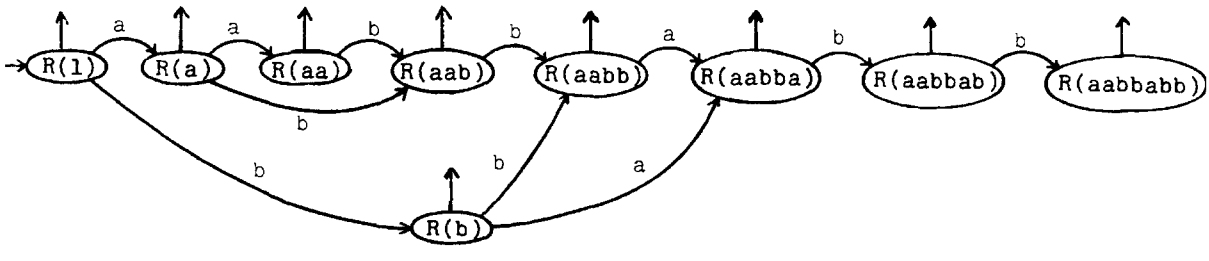


Fig. 1. (Minimal) factor automaton for *aabbabb*.

Lemma 1.1 summarizes two basic properties of the relation R.

1.1. Lemma. *Let* $x \in A^*$ *and* $y, z, u \in F(x)$. *Then*:

(i) $yRz \Rightarrow (y \in S(z)$ *or* $z \in S(y))$;

(ii) $(y \in S(u)$ *and* $u \in S(z)$ *and* $yRz) \Rightarrow uRz$.

Proof. (i) Let w be the longest word such that $yw \in F(x)$. Since yRz, w is also the longest word such that $zw \in F(x)$. Then, $yw \in S(x)$ and $zw \in S(x)$. So, y and z are both suffixes of xw^{-1} which gives the conclusion.

(ii) If $zw \in F(x)$, by $u \in S(z)$ we get $uw \in F(x)$. Conversely, let w be such that $uw \in F(x)$. Since $y \in S(u)$, yw belongs to $F(x)$ and this is also true for zw because y and z are equivalent. This proves uRz. □

The minimal deterministic automaton $\mathcal{S}(x)$ which recognizes $S(x)$, the set of suffixes of x, could be defined in the same way. From an algorithmic point of view, the construction of $\mathcal{S}(x)$ may easily be done from $\mathcal{F}(x\$)$, the factor automaton of x followed by a marker \$ not in A. This results from the equality

$$S(x) = (F(x\$) \cap A^*\$)\$^{-1},$$

which means that suffixes of x are precisely those factors of $x\$$ that are followed by the marker \$. Then, if $\mathcal{F}(x\$)$ is given, $\mathcal{S}(x)$ is built by making terminal those states of $\mathcal{F}(x\$)$ on which a \$-transition is defined and by deleting all \$-transitions together with the state corresponding to the equivalence class $R_{x\$}(x\$)$.

The number of states of $\mathcal{S}(x)$ is at least that of $\mathcal{F}(x)$ and the relation between these two numbers is established in Section 7.

Figure 2 shows an example of a suffix automaton.

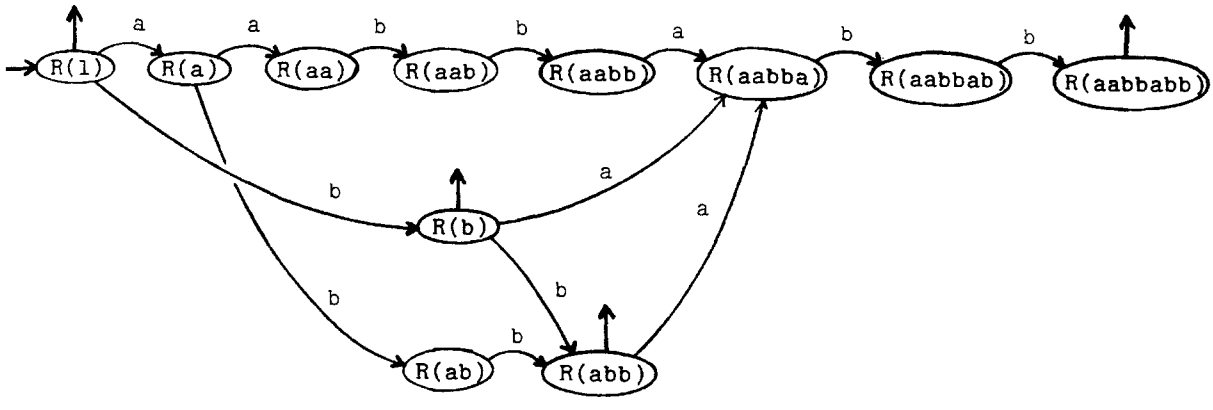


Fig. 2. (Minimal) suffix automaton for *aabbabb*.

2. Factor transducers

In string-matching or related problems, the information (the word is a factor or not) given by an automaton is often not sufficient, and a position of the searched word is also needed. The natural way to deal with that problem is to consider (sequential) transducers instead of automata. The output of the transducer is required to specify which occurrence of the input word has been found.

Different kinds of transducers associated to left sequential functions can be considered, but we are mainly interested in those transducers for which the underlying automaton is minimal.

The word x still being fixed, our first example is the transducer associated to the left sequential function p_x (or p) called *prefix function*. It is defined from $F(x)$ to $P(x)$, the set of prefixes (left factors) of x, by

$$p(y) = \text{shortest } w \in A^* \text{ s.t. } \exists u, v \in A^*,\ w = uy \text{ and } wv = x.$$

Figure 3 shows the transducer associated with the prefix function on *aabbabb*. On input *bba*, for instance, this transducer outputs *aabba* which is $p(bba)$ in the word *aabbabb*.

The reason why the transducer associated with p can be built upon the minimal factor automaton $\mathcal{F}(x)$ is given by the next lemma.

2.1. Lemma. *Let $x \in A^*$ and $y, z \in F(x)$.*

(i) $yRz \Rightarrow p(y) = p(z)$.

(ii) *If $yz \in F(x)$, then $p(y)$ is a prefix of $p(yz)z^{-1}$.*

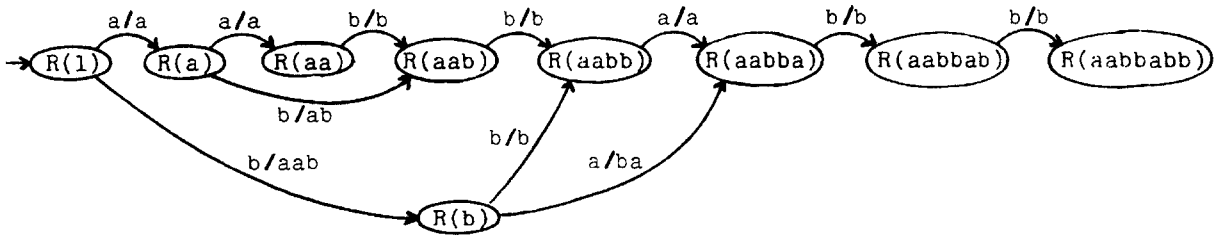


Fig. 3. Transducer associated with the prefix function on *aabbabb*.

Proof. (i) Note that if w is the longest word in $y^{-1}F(x)$, then $p(y)w = x$. By yRz, with the same word w we have $p(z)w = x$ and therefore $p(y) = p(z)$.

(ii) If $p(yz) = uyz$, by definition of p, $p(y)$ is prefix of uy which is $p(yz)z^{-1}$. □

Lemma 2.1 allows us to associate outputs to transitions in $\mathcal{F}(x)$. Namely, if $R(y).a$ is defined in $\mathcal{F}(x)$, the corresponding output is equal to $p(y)^{-1}p(ya)$ which is well-defined since $p(y)$ is a prefix of $p(ya)$.

We are in fact more interested in another kind of transducer, called the factor transducer. The *factor transducer* $\mathcal{C}(x)$ of a word x has $\mathcal{F}(x)$, the minimal factor automaton for x, as its underlying automaton. Its output, on a factor y of x, is the position of the first occurrence of y in x. In other words, the factor transducer for x is associated with the left sequential function pos from $F(x)$ to $\mathbb{N}$ defined by (for $y \in F(x)$)

$$\text{pos}(y) = |p(y)| - |y|.$$

For the transducer $\mathcal{C}(x)$, the output corresponding to a transition in $\mathcal{F}(x)$, $R(y).a$, is denoted $R(y)*a$ and is equal to $|p(ya)| - |p(y)| - 1$. As above, Lemma 2.1 ensures that this definition is coherent. Figure 4 shows the factor transducer for *aabbabb*. It outputs 2 $(=2+0+0)$ on input *bba*; this means that a prefix of x of length two (aa) precedes the first occurrence of *bba* in *aabbabb*.

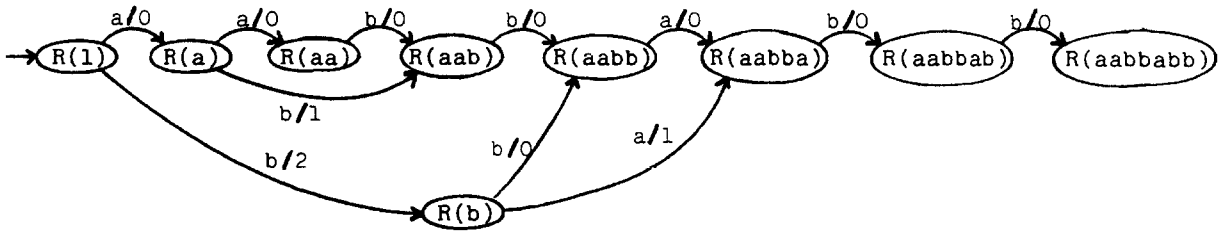


Fig. 4. Factor transducer for *aabbabb*.

It is worth noting that the noncommutative version of the function pos, which returns $p(y)y^{-1}$ (note that y is a suffix of $p(y)$), leads to a transducer that may have more states than $\mathcal{F}(x)$. The word *ababb* is an example of such a phenomenon.

3. Suffix links

Our construction of the factor transducer $\mathcal{C}(x)$ lies heavily on a function defined on states of $\mathcal{C}(x)$ and which is called a suffix link. The situation is analogous to

the construction of the minimal deterministic automaton recognizing A^*x, which yields the well-known string-matching algorithm designed by Knuth, Morris and Pratt [11]. In this case, they consider two kinds of what they call failure functions.

Given a word x in A^*, the *suffix link* s_x (or s) is defined as a function from $F(x)-\{1\}$ to $F(x)$ by

$$s(y) = \text{longest } w \in S(y) \text{ such that not } wR_x y.$$

This definition is quite natural in terms of automata where default state functions are used to implement automata efficiently [3].

Suffix links can also be used to prove the linearity of the size of $\mathscr{C}(x)$. We first prove some properties of s.

3.1. Lemma. *Let $x \in A^*$ and $y, z \in F(x)$.*

(i) $yRz \Rightarrow s(y) = s(z)$.

(ii) *$s_x(x)$ is the longest suffix of x that occurs twice (at least) in x.*

(iii) *Not $yRp(y) \Rightarrow \exists u$, $v \in F(x)-\{1\}$, $s(u) = s(v) = y$ and not uRv.*

Proof. (i) By Lemma 1.1(i), it may be assumed that yRz and $y \in S(z)$. Since $s(z)$ is not equivalent to z, by Lemma 1.1(ii), y cannot be a suffix of $s(z)$. Therefore, $s(z)$ is equivalent to the longest suffix of y which is not equivalent to z and y. So, $s(z)$ is $s(y)$.

(ii) Let w be the longest word such that $s(x)w \in F(x)$. Since $s(x)$ is not equivalent to x, w is nonempty. So, $s(x)$ occurs twice in x, at the end of x and before w. If v is a suffix of x that occurs twice, then the longest word w such that $vw \in F(x)$ must be nonempty and so v is not equivalent to x. Thus, v is shorter than (or of the same length as) $s(x)$.

(iii) Assume y is a factor of x which is not R-equivalent to $p(y)$. Since not $yRp(y)$, one may consider the shortest word u such that

$$y \in S(u),\ u \in S(p(y)) \text{ and not } yRu.$$

Note that $u \neq 1$ and that $s(u) = y$.

Now, since u and y are not equivalent, they must occur in different right contexts. This means (since $y \in S(u)$) that there exists a $w \in S(x)$ such that $yw \in S(x)$ and $uw \notin S(x)$. Then, let v be the shortest word such that

$$y \in S(v),\ v \in S(xw^{-1}) \text{ and not } yRv.$$

Once more we have $v \neq 1$ and $s(v) = y$. We also have not uRv since w distinguishes these two words. □

Property (i) in Lemma 3.1 allows to define s on states of $\mathscr{C}(x)$ (except on state $R(1)$). Property (ii) gives an interesting characterization of $s(x)$ which is to be put in parallel with the failure function of [16]: the longest proper suffix of x which is

also a prefix of x. This contrasts with the suffix link defined by McCreight [15] in his suffix-tree construction: $s(x)$ is the longest proper suffix of x.

Property (iii) in Lemma 3.1 helps proving a first upper bound on the number of states of $\mathscr{C}(x)$. Note that since $|s(y)|<|y|$, the suffix link s provides a tree-structure for the set of states of $\mathscr{C}(x)$.

3.2. Proposition ([7]). *Let $x\in A^*$ and let $e(x)$ be the number of states of $\mathscr{C}(x)$.* Then

$$|x|+1\leqslant e(x)\leqslant 2|x|+1.$$

Proof. By Lemma 3.1, the set of states of $\mathscr{C}(x)$ may be considered as a tree for which s is the function 'father'. The root of the tree is $R(1)$. Now, transform this tree in a complete tree as follows:

(a) replace each node without son by a leaf labelled by the corresponding prefix of x;

(b) to each node labelled by the R-equivalence class of a prefix of x and which has at least one son, add as a new son a leaf labelled by the prefix.

The fact that nodes without son, in the initial tree, are labelled by the equivalence class of a prefix of x is a consequence of Lemma 3.1(iii).

The complete tree obtained after rules (a) and (b) have been applied has exactly $|x|+1$ leaves and each of its internal nodes has at least two sons (by Lemma 3.1(iii) again).

The maximum number of nodes in the complete tree is then achieved when the tree is binary. It has $2|x|+1$ nodes, which proves the upper bound.

The states encountered during reading x in $\mathscr{C}(x)$ is a set of pairwise distinct states. Its cardinality is then $|x|+1$ which gives the lower bound. □

Figure 5 gives an example of a suffix function and its associated complete tree.

As we shall see in the next section, the upper bound on $e(x)$ given in Proposition 3.2 can be reduced. The bounds on the number of transitions in $\mathscr{C}(x)$ given in the next proposition are the best possible.

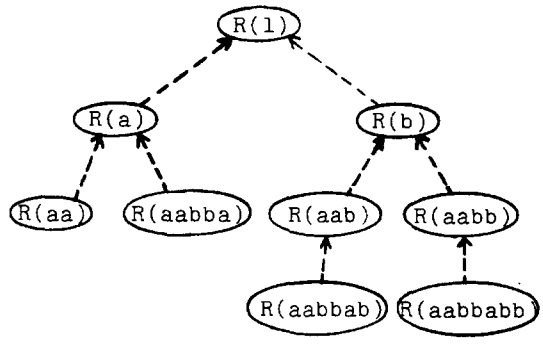
3.3. Proposition ([7]). *Let $x\in A^*$ and let $t(x)$ be the number of transitions defined in $\mathscr{C}(x)$. Then*

$$|x|\leqslant t(x)\leqslant e(x)+|x|-2.$$

Proof. As a labelled graph, $\mathscr{C}(x)$ is connex and then the minimum number of edges is $|x|$.

Consider a spanning tree for $\mathscr{C}(x)$ which has a branch labelled by x. The number of edges in the tree is $e(x)-1$.

To count the extra transitions (not in the spanning tree), we associate to each such transition (by a from q to q') a suffix yaz of x as follows: y is the label of the



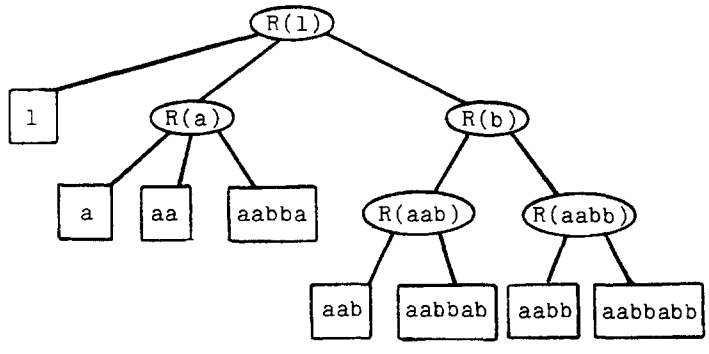


Fig. 5. Suffix function for *aabbabb* and associated complete tree.

path in the spanning tree from the initial state to q and z lengthens ya in a suffix of x. The correspondence is one-to-one and words 1 and z are not reached. So, the maximum number of extra edges is $|x|-1$ which gives the upper bound. □

4. Bounds on the size of factor transducers

The upper bound given on the number of states of factor transducers in the previous section is not tight. To refine this bound we look more precisely at how $\mathscr{C}(xa)$ is built from $\mathscr{C}(x)$. This has several advantages. First, the next theorem on size of $\mathscr{C}(xa)$ related to the size of $\mathscr{C}(x)$ gives a base for the proof of our on-line construction of factor transducers. Second, it clarifies the link between minimal factor automata and minimal suffix automata.

Before stating our main theorem, we need one more definition. Let $x \in A^*$ and let u be $s_x(x)$. It is known by Lemma 3.1 that u is the longest suffix of x that occurs twice in x. In the transducer $\mathscr{C}(x)$, this means that the paths labelled by x and u and starting at the initial state leads to two distinct states. During the construction of $\mathscr{C}(xa)$, it happens that states of a part of the path labelled by u must be duplicated. The other part of the path is then said to be *safe* (see Fig. 6).

Formally, safe(x) is defined to be the longest prefix of u ($= s_x(x)$) which is length maximal in its R_x-equivalence class.

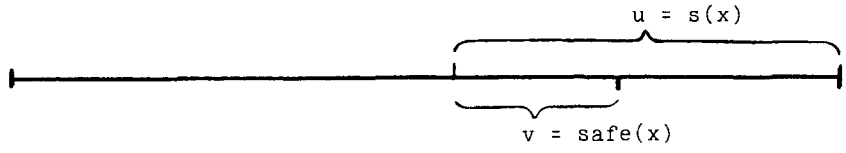


Fig. 6. Suffix link u and its safe part v.

4.1. Theorem. *Let $x \in A^+$ and $a \in A$. Let u be the suffix link $s(x)$ of x and v be* safe(x). *Then $(e(x)$ is the number of states of the factor transducer $\mathscr{C}(x))$:*

$$e(xa) = e(x)+1 \qquad \text{if } ua \in F(x),$$

$$e(xa) = e(x)+|v^{-1}u|+1 \qquad \text{otherwise.}$$

Proof. It is first shown that R_{xa} is a refinement of R_x. Let y and z be such that $yR_{xa}z$. Consider any w in A^* for which $yw \in F(x)$. Then, there exists a $w' \in A^*$ such that $yww'a \in F(xa)$. Words y and z being R_{xa}-equivalent, $zww'a$ also belongs to $F(xa)$ and, therefore, $zw \in F(x)$. So, yR_xz holds.

We now turn to the proof.

Case 1: $ua \in F(x)$. Only one R_x-equivalence class, namely $\{y \in A^* | y \notin F(x)\}$, splits in exactly two R_{xa}-subclasses:

$$\{y \in A^* | y \notin F(xa)\} \quad \text{and} \quad S(xa) - F(x).$$

The latter class is also

$$\{wa \in S(xa) | |w| > |u|\},$$

by the definition of u, by Lemma 3.1(iii) and the hypothesis $ua \in F(x)$. Transducer $\mathscr{C}(xa)$ has only one more state than $\mathscr{C}(x)$.

Case 2: $ua \notin F(x)$. Again a new state arises in $\mathscr{C}(xa)$ from the R_{xa}-equivalence class $S(xa) - F(x)$.

We examine the R_x-equivalence classes that yield several R_{xa}-subclasses. Let w and w' be such that wR_xw' but not $wR_{xa}w'$. By Lemma 1.1, it may be assumed that $w \in S(w')$ and that w' is length maximal in its R_x-class. So, by hypothesis, there exists a $z \in A^*$ satisfying $wza \in S(xa)$, $w'za \notin F(xa)$, and $w'z \in F(x)$. Let z be the longest word with these conditions.

The word $w'z$ cannot be a suffix of x since $w'za \notin F(xa)$. Thus, wz (which is a suffix of $w'z$) is a suffix of x and occurs twice in x. From the definition of u and Lemma 3.1(ii) it follows that wz is a suffix of u.

Let yw be the longest suffix common to w' and uz^{-1}. It is shown that yw and w are R_{xa}-equivalent and that yw is a prefix of u of length greater than v.

By Lemma 1.1(ii), yw is R_x-equivalent to w. Assume, ab absurdo, that yw and w are not R_{xa}-equivalent. Then, there exists a longest word z' such that $wz' \in S(u)$ and $ywz' \in F(x) - S(u)$. Let $z'' \in A^+$ be such that $uz'' \in S(x)$; word z'' exists because

u occurs twice in x. But now $wz'z'' \in S(x)$ occurs also twice in x because ywR_xw and $ywz'z'' \in S(x)$, so $wz'z''$ is a suffix of u which contradicts the maximality of z'. Thus, $ywR_{xa}w$.

If yw is not a prefix of u, then, for two distinct letters b and c, byw is a suffix of w' and cyw is in $F(u)$. The word $ywzz''$ which is a suffix of x occurs twice because $w'R_xyw$, $cywzz'' \in S(x)$, and $cyw \notin S(w')$. But, this contradicts the maximality of z.

Finally, the R_x-equivalence class of w splits in exactly two R_{xa} subclasses: the subclass of suffixes of w' that are not R_{xa}-equivalent to w' and the other suffixes of w' including of course w' itself. It is important to note that the first subclass contains those suffixes of w' which are in $F(u)$, and that all these words are R_{xa}-equivalent to the above prefix yw of u. The word yw is not length maximal in the R_x-class and then $|yw| > |v|$ by the definition of v. Note that yw is length maximal in its new R_{xa}-equivalence class.

The number of R_x-equivalence classes split in two R_{xa}-subclasses is then $|v^{-1}u|$ which concludes the proof. □

4.2. Corollary. *The number $e(x)$ of states of the factor transducer $\mathscr{C}(x)$ of a word x in A^* satisfies:*

$$\text{if } |x| \leqslant 3, \quad e(x) = |x| + 1,$$

$$\text{if } |x| > 3 \quad |x| + 1 \leqslant e(x) \leqslant 2|x| - 2 \text{ and}$$

$$e(x) = 2|x| - 2 \text{ iff } x \in ab^*c,\ a \neq b,\ b \neq c.$$

Proof. The case where $|x| \leqslant 3$ can easily be checked by hand. Consider a word xc ($x \in A^*$ and $c \in A$) with $|xc| > 3$. Let u be $s(x)$ and v be safe(x).

If $uc \in F(x)$ or $v = u$, by Theorem 4.1 and the induction hypothesis one gets

$$e(xc) = e(x) + 1 \leqslant 2|x| - 2 + 1 < 2|xc| - 2,$$

which additionally shows that the upper bound is strict.

If $uc \notin F(x)$ and $v \neq u$, Theorem 4.1 and its proof give

$$e(xc) = e(xu^{-1}v) + 2|v^{-1}u| + 1.$$

By the induction hypothesis, if $|xu^{-1}v| \geqslant 3$, one gets

$$e(xc) \leqslant 2|xu^{-1}v| - 2 + 2|v^{-1}u| + 1 = 2|xc| - 3,$$

and again the upper bound is not reached.

It remains to check what happens when $|xu^{-1}v| < 3$. The only possibility is $|xu^{-1}v| = 2$, since otherwise x would be in a^* (for some a in A), which would contradict $v \neq u$. For the same reason, we also deduce that $v = 1$. Then

$$e(xc) = 3 + 2|u| + 1 = 2|x| = 2|xc| - 2.$$

The upper bound is reached. The word xu^{-1} is aa or ab (for a, b in A and $a \neq b$). The former case implies $u \in a^*$ which is impossible. If $xu^{-1} = ab$, since u occurs twice in x, we have $u \in (ab)^*$, $u \in (ab)^*b$ or $u \in b^*$. The latter case is the only possibility which gives $x \in ab^*$, and since $uc \notin F(x)$ we also have $c \neq b$. □

4.3. Corollary. *The number $t(x)$ of transitions in $\mathscr{C}(x)$, the factor transducer of x in A^*, satisfies:*

$$\text{if } |x| \leq 3, \quad |x| \leq t(x) \leq 2|x| - 1,$$

$$\text{if } |x| > 3, \quad |x| \leq t(x) \leq 3|x| - 4 \text{ and}$$

$$t(x) = 3|x| - 4 \text{ iff } x \in ab^*c,\ a \neq b,\ b \neq c,\ c \neq a.$$

Proof. Bounds come directly from those of Proposition 3.3 and Corollary 4.2. When $|x| > 3$, the upper bound can only be reached when $e(x)$ is maximal, that is, when $x \in ab^*c$, with $a \neq b$ and $b \neq c$. If $x \in ab^*a$, $t(x)$ is equal to $3|x| - 5$, while, if $x \in ab^*c$ with $c \neq a$, we get the maximum number of transitions $3|x| - 4$. □

Figure 7 gives an example of a biggest factor transducer.

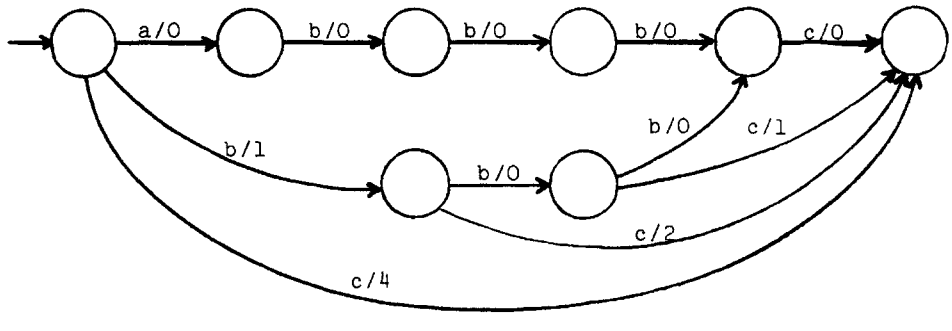


Fig. 7. A biggest factor transducer.

5. Construction of factor transducers

Our algorithm, given in Fig. 8, which builds the minimal factor transducer $\mathscr{C}(x)$ of a word x in A^*, follows Theorem 4.1 and its proof. It processes the word on-line and its structure is close to Knuth, Morris and Pratt's algorithm, especially in the computation of suffix links through the function 'suffix' which is written apart.

We now describe the meaning of the variables used in the algorithm. While doing this, it is assumed that x is the prefix of the input word which has just been processed. The letters of x are the a_i's and n is its length:

$$x = a_1 \ldots a_n.$$

Let u be $s(x)$ and v be safe(x). The integer m is such that

$$v^{-1}u = a_{m+1} \ldots a_n.$$

```
begin create state art; l(art) ← p(art) ← −1;
  create new state init; l(init) ← p(init) ← 0;
  last ← init; s(init) ← art;
  m ← n ← 0; r ← art; r' ← init;

  while input is not empty do
    read next letter a; art.a ← init;
    create new state q; l(q) ← l(last) + 1; p(q) ← p(last) + 1;
    last.a ← q; last * a ← 0;

    if s(last).a defined then s(q) ← s(last).a
    else while m < n do
           m ← m + 1; b ← x_m; r' ← r'.b;
           create copy r̄ or r.b
             with same transitions and attributes;
           l(r̄) ← l(r) + 1; s(r.b) ← r̄; s(r') ← r̄;
           r.b ← r̄; r * b ← p(r̄) − p(r) − 1;
           s(r̄) ← suffix(r, b).b;
           r ← r̄
         endwhile;
         r ← suffix(last, a); s(q) ← r.a
    endif;

    if r' = last and l(r.a) = l(r) + 1
    then m ← m + 1; r ← r.a.; r' ← r'.a endif;
    last ← q; n ← n + 1; x_n ← a
  endwhile
end.

function suffix(r, b);
  if s(r).b undefined or l(s(r).b) ≥ l(r.b)
  then s(r).b ← r.b; s(r) * b ← p(s(r).b) − p(s(r)) − 1;
       return(suffix(s(r), b))
  else return(s(r)) endif
endfunction.
```

Fig. 8. Construction of factor transducers.

Special states of $\mathscr{C}(x)$ are named init, last, art, r and r', with the following meaning:

- init is the initial state of $\mathscr{C}(x)$,
- last is init.x, and
- art is an artificial state not actually in $\mathscr{C}(x)$.

It is assumed that art.a = init for each letter a in A. State art acts as a list header and is used as a sentinel by the function suffix. Once the transducer has entirely been built, 'art' may be thrown out.

- r marks the end of the safe path from init and is init.v,
- r' corresponds to r and is init.$(xu^{-1})v$ which is also init.$a_1 \ldots a_m$.

To each state q of $\mathscr{C}(x)$ are associated three attributes, $p(q)$, $s(q)$, and $l(q)$. If it is assumed that w is any word such that init.$w = q$, then:

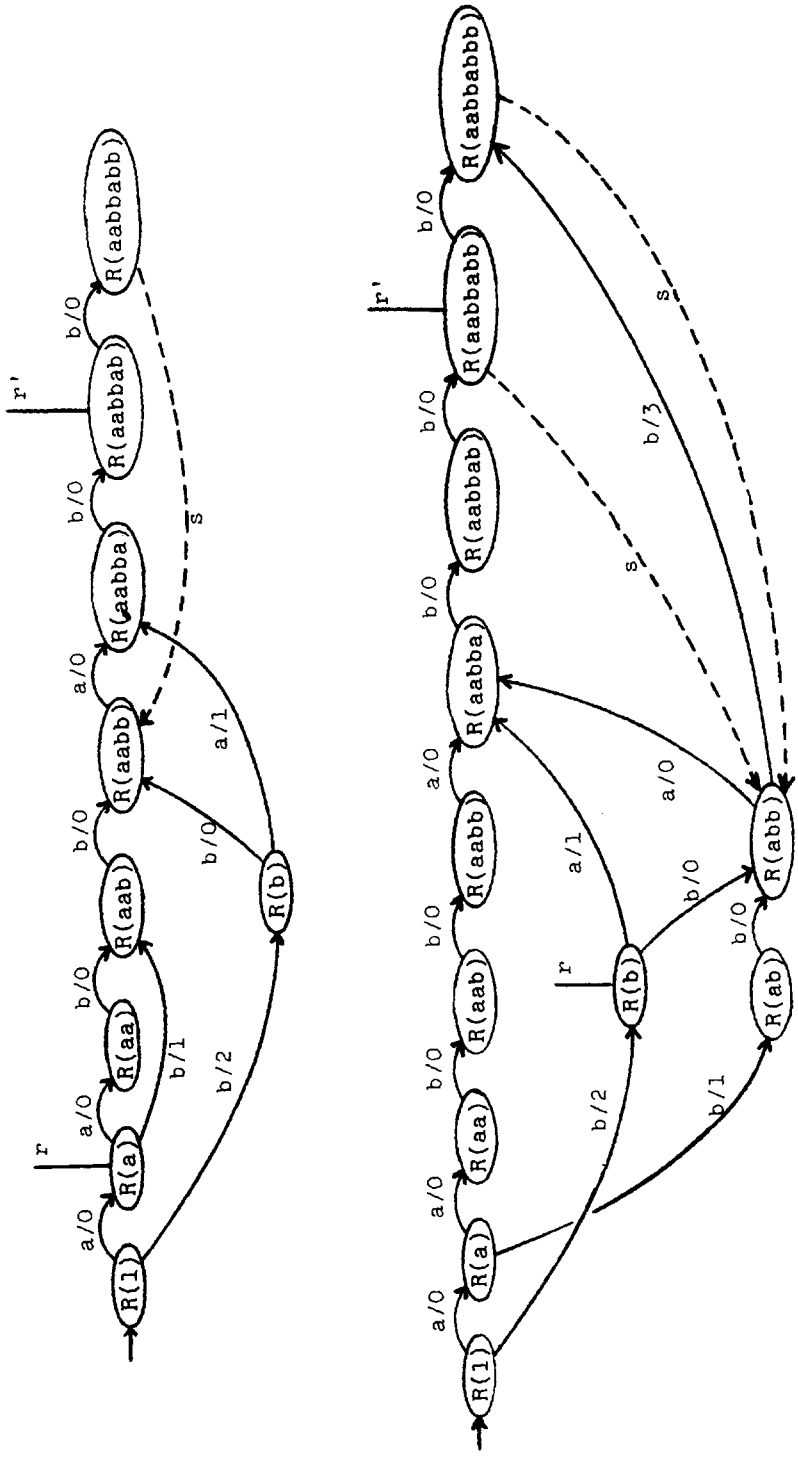


Fig. 9. One step in the algorithm: from $\mathcal{C}(aabbabb)$ to $\mathcal{C}(aabbabbb)$.

- $p(q)$ is the length of $p(w)$ which is independent of w by Lemma 2.1,
- $s(q)$ is the state init.$s(w)$ which is independent of w by Lemma 3.1,
- $l(q)$ is the length of the longest w such that init.$w = q$.

After $\mathscr{C}(x)$ has been built, a new letter a is read on the input. A first state q is created and linked to last; it will become the 'last' state at the next step. Now, the only factors of xa that might not be recognized by the transducer are suffixes of $s(x)a$ and in this situation $s(x)a$ itself is not recognized. So, after a test according to Theorem 4.1, other new states are possibly created.

The aim of the function suffix is to help calculate the suffix link of xa or more exactly of $q = \text{init}.xa$. This function is also used to compute the new suffix links of duplicated states.

Given a state r and a letter b, suffix returns the state $s^k(r)$ for the least integer $k \geqslant 1$ such that $s^k(r).b$ is defined and is not equivalent to $r.b$. This latter condition is checked with the help of attribute l. For doing so, suffix goes through the suffix links until the condition is satisfied. The existence of state art ensures us that suffix stops. While suffix works some transitions may have to be redefined.

In terms of words, if w is the longest suffix of x such that wa occurs twice in xa, then suffix returns the state init.w.

The main **while**-loop of the algorithm ends with a conditional recomputation of r and r' which are associated to xa. The next lemma shows why this test is particularly simple.

5.1. Lemma. *Let $x \in A^*$ and u be the suffix link $s_x(x)$ of x. If $ua \notin F(x)$ and wa is longest suffix of xa such that $wa \in F(x)$, then:*
- *w is length maximal in its R_{xa}-equivalence class, and*
- safe(xa) *is equal to w or wa.*

Proof. This is mainly a re-statement of the remark at the end of the proof of Theorem 4.1.

Since wa occurs twice in xa, w occurs twice in x and is then a suffix of u. Furthermore, $ua \notin F(x)$ and $wa \in F(x)$ imply that w is a proper prefix of u, which means that $bw \in S(u)$ for some b in A.

Assume now that w' is R_{xa}-equivalent to w and $|w'| > |w|$. By Lemma 1.1, there is no restriction in considering $w' = cw$ for some c in A. Since $wa \in F(x)$, there exists a $y \in A^*$ such that $waya \in S(xa)$ and thus $cwaya \in F(xa)$. But the length hypothesis on w implies $cwa \notin S(xa)$ and thus $cw \notin S(u)$. So, $b \neq c$.

By the definition of u, there must exist $z \in A^+$ such that $uz \in S(x)$ and thus $bwz \in S(x)$ or $bwza \in S(xa)$. By $(w, cw) \in R_{xa}$, we get $cwza \in F(x)$ which gives a contradiction with the definition of w since $|wz| > |w|$. □

Lemma 5.1 is not actually necessary to give the linear time complexity of the algorithm in Fig. 8. But it brings to the algorithm all its simplicity, and makes it very easy to design (see also Fig. 9).

6. Implementation and complexity

Implementation of transducers needs to be specified before discussing the complexity of the algorithm of Fig. 8 which builds a factor transducer.

Each state q of transducers may be seen as a block of information which contains $p(q)$, $s(q)$, $l(q)$ and a pointer to the list of transitions defined on q. This list then contains triples (a, q', i) such that $q.a = q'$ and $q * a = i$. A state may be identified with an address and we assume that accesses to attributes p, s, or l are realized in constant time. The transducer being deterministic, to a state q is associated at most one triple (a, q', i) for a given letter a, and the list of transitions defined on q has length at most $|A|$. Using standard techniques (search trees, hashed tables, etc.) to implement these lists leads to an $O(\log|A|)$ time complexity to access, define or redefine $q.a$ and $q * a$. We are now ready to prove the linearity of the construction.

6.1. Theorem. *The construction of the (minimal) factor transducer $\mathscr{C}(x)$ of a word x in A^* by the algorithm in Fig. 8 takes $O(|x|\log|A|)$ time and $O(|x|)$ space.*

Proof. The space used by the algorithm is proportional to the size of $\mathscr{C}(x)$ and variables such as art, init, last, q, r, ... Applying Corollaries 4.2 and 4.3, we get the space complexity $O(|x|)$.

Note that each time instructions in either **while**-loop are executed, a state is created. Each instruction in the **while**-loops, except the calls to 'suffix', takes a constant time or $O(\log|A|)$ time in the worst case. So, if calls to 'suffix' are eliminated for a moment, the total cost of other instructions is $O(|x|\log|A|)$ (applying again Corollary 4.2 on the number of states of $\mathscr{C}(x)$).

Each nonrecursive call to the 'suffix' function contributes $O(\log|A|)$ to the total cost of instructions of the **while**-loops. Then, for the same reason as above, the aggregate cost of all nonrecursive calls to suffix is $O(|x|\log|A|)$.

Consider now the recursive calls to 'suffix'. Each takes $O(\log|A|)$ time. The reason why their total cost is $O(|x|\log|A|)$ is that each recursive call to suffix strictly shortens the suffix link (except maybe the last call when it returns state 'art'). In terms of words, if $y = a_1a_2 \ldots a_ka_{k+1} \ldots a_n$ is the already processed word and if $s_y(y) = a_{k+1} \ldots a_n$, then each recursive call to 'suffix' strictly increases index k. The total number of these calls is then bounded by $|x|$. This concludes the proof. □

6.2. Corollary. *On a given alphabet A, factor transducers $\mathscr{C}(x)$ or minimal factor automata $\mathscr{F}(x)$ (where x is a word in A^*) can be built in time linear in the length of x.*

7. Suffix transducers

This section deals with suffixes instead of factors. We have already mentioned in Section 1 how the suffix automaton $\mathscr{S}(x)$ can be built from $\mathscr{F}(x)$. In fact, construction

of suffix automata can be made easier as we shall see here. The algorithm given in [7] builds an automaton which has exactly the same number of states as $\mathcal{S}(x)$.

Apart from the number of states, another element distinguishes $\mathcal{S}(x)$ from $\mathcal{F}(x)$: the terminal state. In $\mathcal{F}(x)$ or $\mathcal{C}(x)$ all states are terminal since minimal automata without 'sink' state are considered. For suffix automata not all states are terminal, as shown in Fig. 2.

The natural function which defines a position of a suffix y in a word x is

$$f_x(y) = |xy^{-1}|.$$

This function from $\mathcal{S}(x)$ to $\mathbb{N}$ is no longer sequential as 'pos' is, for a suffix of x can also happen to occur inside x. But, if f is written as

$$f_x(y) = \mathrm{pos}_x(y) + (|x| - |p(y)|),$$

it becomes clear that f_x is represented by a subsequential transducer (the quantity $|x| - |p(y)|$ depends on the R-equivalence class of y by Lemma 2.1). In addition to the outputs associated with transitions, each terminal state q bears an extra output, denoted out(q), which is $(|x| - |p(y)|)$ where y is a word of the R-equivalence class q. We call *suffix transducer*, $\mathcal{C}_s(x)$, of a word x the subsequential transducer representing f_x and whose underlying automaton is the minimal suffix automaton $\mathcal{S}(x)$.

Two examples of suffix transducers are given in Figs. 10 and 11. Reading *abb* in $\mathcal{C}_s(aabbabb)$ we reach a terminal state and the total output is $0+1+0+3=4$ which is the length of *aabb* prefix of *aabbabb* before its suffix *abb*.

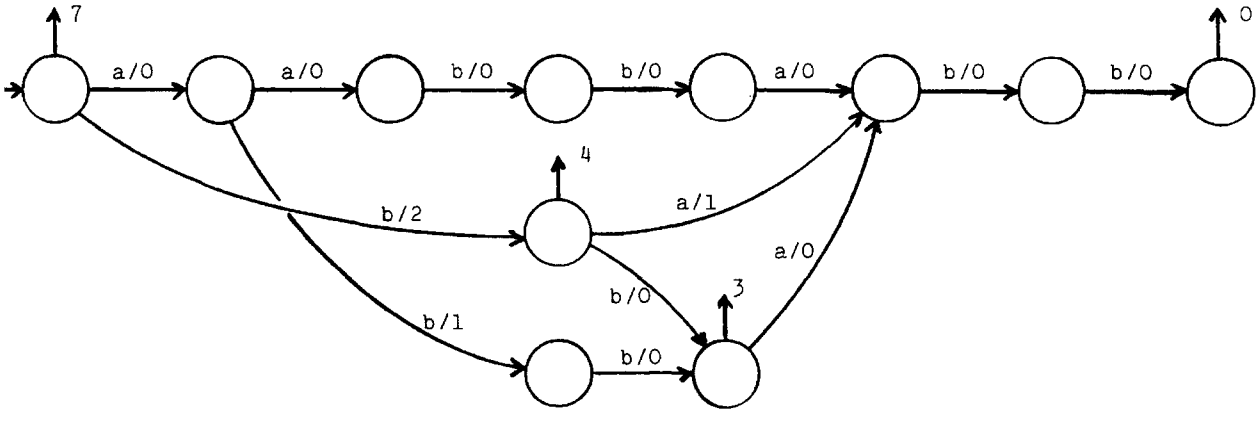


Fig. 10. Suffix transducer of *aabbabb*.

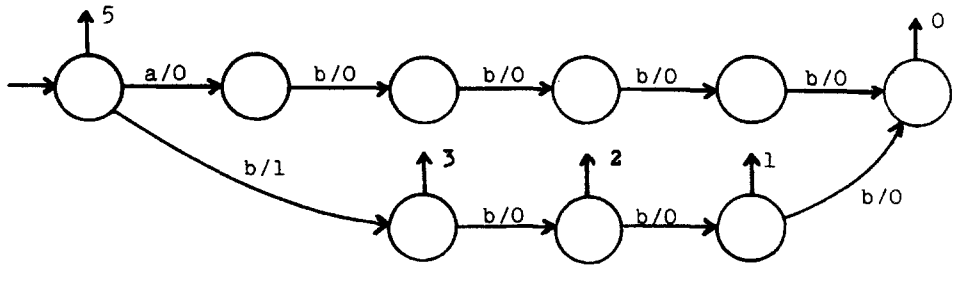


Fig. 11. Suffix transducer of *abbbb*.

The remark on suffix automata in Section 1 together with Theorem 4.1 lead to a relation between the numbers of states of $\mathscr{C}(x)$ and $\mathscr{C}_s(x)$ for a given word x.

7.1. Proposition. *Let $x \in A^+$. Let u be the suffix link $s(x)$ of x and v be* safe(x). *Then, the number $e_s(x)$ of states of suffix transducer $\mathscr{C}_s(u)$ is $e(x)+|v^{-1}u|$.*

With the help of Corollary 4.2, optimal bounds on $e_s(x)$ are obtained.

7.2. Corollary. *The number $e_s(x)$ of states of the suffix transducer $\mathscr{C}_s(x)$ of a word x in A^* satisfies*:

$$\text{if } |x| \leqslant 2, \quad e_s(x) = |x| + 1,$$

$$\text{if } |x| > 2, \quad |x| + 1 \leqslant e_s(x) \leqslant 2|x| - 1 \text{ and}$$

$$e_s(x) = 2|x| - 1 \text{ iff } x \in ab^*,\ a \neq b.$$

The proofs of Proposition 7.1 and Corollary 7.2 are straightforward once we have noted that $e_s(x) = e(x\$) - 1$ if "\$" is a marker (not in A).

Figure 11 gives an example of the maximum number of states for a word of length 5.

It is also possible to give optimal bounds on the number of transitions in $\mathscr{C}_s(x)$.

7.3. Proposition. *The number $t_s(x)$ of transitions of the suffix transducer $\mathscr{C}_s(x)$ of a word x in A^* satisfies*:

$$\text{if } |x| \leqslant 3, \quad |x| \leqslant t_s(x) \leqslant 2|x| - 1,$$

$$\text{if } |x| > 3, \quad |x| \leqslant t_s(x) \leqslant 3|x| - 4 \text{ and}$$

$$t_s(x) = 3|x| - 4 \text{ iff } x \in ab^*c,\ a \neq b,\ b \neq c,\ c \neq a.$$

Proof. We only proof the upper bound for words of length greater than 3. Let d be the number of \$-transitions in $\mathscr{C}(x\$)$. Then, $t_s(x) = t(x\$) - d$.

Assume first that $t(x\$)$ is maximal. By Corollary 4.3, we know that $t(x\$) = 3|x\$| - 4$ and that $x = ab^n\$$ for some $n > 2$. It may be checked that, for words $ab^n\$$, the number d is $n+1$. Thus, in that case, we get

$$t_s(x) \leqslant 3|x| - 5.$$

Assume now that d is minimum. Since x is not empty, this means $d = 2$, $s(x) = 1$, and $t(x) = t_s(x)$. The maximum is reached when $t(x)$ is also maximum, that is, by Corollary 4.3, exactly when $x \in ab^*c$ (a, b, c pairwise distinct letters), because x satisfies $s(x) = 1$ in that case. □

While the construction of $\mathscr{C}_s(x)$ may be done from that of $\mathscr{C}(x\$)$, the algorithm becomes simpler and easier to design. The algorithm for the construction of the (minimal) suffix transducer corresponding to function f_x is given in Fig. 12.

```
begin create state art; l(art) ← p(art) ← −1;
  create new state init; l(init) ← p(init) ← 0;
  last ← init; s(init) ← art;

  while input is not empty do
    read next letter a; art.a ← init;
    create new state q; l(q) ← l(last) + 1; p(q) ← p(last) + 1;
    last.a ← q; last * a ← 0;

    if s(last).a defined then r ← s(last)
    else r ← suffix(last, a) endif;

    if l(r.a) > l(r) + 1 then
      create copy r̄ of r.a. with same transitions and attributes;
      l(r̄) ← l(r) + 1; s(r.a) ← r̄;
      r.a ← r̄; r * a ← p(r̄) − p(r) − 1;
      s(r̄) ← suffix(r, a).a;
    endif;
    s(q) ← r.a.; last ← q
  endwhile;
  q ← last; out(last) ← 0;
  repeat q ← s(q); out(q) ← p(last) − p(q)
  until q = init
end.
```

Fig. 12. Construction of suffix transducers.

The **while**-loop in this algorithm is shorter than in the algorithm of Fig. 8 since we do not have to maintain the path labelled by the unsafe part of the suffix link of word x. When the last transition along the path from initial state labelled by the suffix link becomes unsafe (condition $l(r.a) > l(r) + 1$), then an extra state is immediately created.

At the end of the algorithm, outputs on terminal states are computed during a climbing up the suffix links from the 'last' state. It is as if a new letter or a marker have been encountered after the end of word x.

8. Factorizing words and squares

We discuss here an application of factor transducers and the linear time of their construction to the detection of a repetition inside a word. By doing this, we are led to introduce a particular but quite natural factorization of words which is close to and more efficient than the one introduced by Ziv and Lempel [20] for their data compression method. The problem we are interested in is finding a square in a word. Recall that a *square* is a nonempty word of the form uu. Apart from the naive $O(n^3)$ algorithm, the use of Morris and Pratt's algorithm [16] yields an obvious $O(n^2)$ algorithm on a word of length n. A divide-and-conquer approach gives an

$O(n \log n)$ algorithm which is optimal if the size of the alphabet is not bounded [13]. Thanks to factor transducers we get an $O(n)$ algorithm for finding a square in a word of length n on a fixed alphabet.

Let $x = a_1 \dots a_n$ be a word in A^+. The *f-factorization* $(v_1, \dots, v_m)$ of x is a sequence of nonempty words defined as follows: assume $v_1, \dots, v_{k-1}$ have been defined and $v_1 \dots v_{k-1} = a_1 \dots a_i$ with $i < n$; let u be the longest prefix of $a_{i+1} \dots a_n$ which occurs twice in $v_1 \dots v_{k-1}u$; then, v_k is a_{i+1} if u is empty, and u otherwise.

Example. The *f*-factorization of *abcacbabcabcaa* is (*a, b, c, a, c, b, abca, bca, a*).

As regards the *f*-factorization of x, it may be noted that v_1 is a_1 and, in general, the first occurrence of a letter gives a v_i of length 1.

Computation of *f*-factorizations is not given itself in this paper. It is part of the algorithm of Fig. 14 which uses the factor transducer of the input word. The aim of the internal **while**-loop is to compute the next term v of the *f*-factorization of $a_1 \dots a_n$ together with the position pos of the first occurrence of v. The successive values of v at the end of the main **while**-loop exactly compose the *f*-factorization of $a_1 \dots a_n$ (unless a square is found and the function returns). It also appears that computing the *f*-factorization takes $O(n \log|A|)$ time ($\log|A|$ comes from the computation of transitions).

To deal with squares, two other functions, left and right, are needed. Left takes a couple of square-free words u and v and returns 'square' exactly when uv contains a square centered in u ($\exists u', v', w_1, w_2, w_2 \neq 1$, $u = u'w_1w_2w_1$, $v = w_2v'$), and 'no square' otherwise. The function right may be defined by

$$\text{right}(u, v) = \text{left}(\tilde{v}, \tilde{u}),$$

where $\tilde{u}$ and $\tilde{v}$ are the mirror images of the square-free words u and v. These functions have been introduced by Main and Lorentz [13] who gave a linear algorithm to compute them.

8.1. Theorem ([13]). *When u and v are square-free words of A^*, $\text{left}(u, v)$ may be evaluated in $O(|u|)$ time.*

The proof of Theorem 8.1 is not given here, but elements to do it can be found in Appendix A.

All ingredients, *f*-factorization and the functions left and right, are present to prove the theorem which leads to a linear square-freeness test.

8.2. Theorem. *Let $x = a_1 \dots a_n$ be a word in A^+. Let $(v_1, \dots, v_m)$ be the f-factorization of x. Then, x contains a square iff $\exists k \in \{2, 3, \dots, m\}$ and*

(a) *two occurrences of v_k overlap, i.e., more precisely,*

$$\text{pos}(v_k) + |v_k| \geq |v_1 \dots v_{k-1}|,$$

or (b) *there exists a square the half right part of which is in $v_{k-1}v_k$, i.e., $v_1, \dots, v_k$ are square-free words and $\text{left}(v_{k-1}, v_k)$ or $\text{right}(v_{k-1}, v_k)$ or $\text{right}(v_1 \dots v_{k-2}, v_{k-1}v_k)$ is 'square'.*

Proof. '*If*': When (a) is satisfied, $v_1 \dots v_{k-1}$ can be written as uw, where $|u| = \text{pos}(v_k)$ and w is a nonempty prefix of v_k. Then, $v_1 \dots v_{k-1}v_k$ contains the square ww. When (b) is satisfied, the definitions of left and right give the answer.

'*Only if*': Let uww be the shortest prefix of x which contains a square ($w \neq 1$). Let k be the smallest index in $\{2, 3, \dots, m\}$ such that uw is a prefix of $v_1 \dots v_{k-1}$. We assume (a) is false up to k and prove condition (b). Note that $v_1, \dots, v_k$ are square-free since $v_1 \dots v_{k-1}$ is square-free and $v_k \in F(v_1 \dots v_{k-1})$. If $v_{k-1}v_k$ is not square-free, then it must contain a square centered in v_{k-1} or in v_k which means that left(v_{k-1}, v_k) or right(v_{k-1}, v_k) is 'square'. Otherwise, definition of v_k insures that uww is a prefix of $v_1 \dots v_k$ and by the minimality of k the middle of ww is in $v_{k-1}v_k$, i.e., right($v_1 \dots v_{k-2}$, $v_{k-1}v_k$) is 'square'. □

Figure 13 represents the two cases of Theorem 8.1. The integers which appear there correspond to the variables pos, i, and j of the function square in Fig. 14. This function works exactly as said in Theorem 8.2. The internal **while**-loop searches for the longest prefix $a_{i+1} \dots a_j$ of $a_{i+1} \dots a_n$ which occurs twice in $a_1 \dots a_j$ using the factor transducer. After that, a first test ($i = j$) eliminates the case where a_{i+1} is a new letter since then no square can be found. Then, conditions (a) ($\text{pos} + (j - i) \geq i$) and (b) are tested in that order.

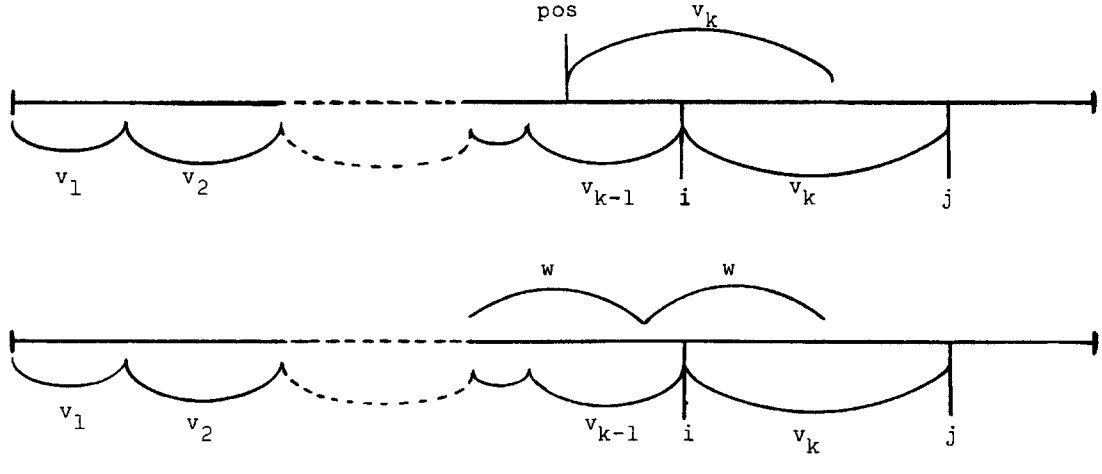


Fig. 13. Cases (a) and (b) of Theorem 8.1.

To make the presentation easier, construction of the factor transducer is not given in the function square, but its construction can be realized as searching for a square goes. The function so becomes almost on-line and finds the smallest k such that $v_1 \dots v_k$ contains a square if there is one.

8.3. Theorem. *Function* square (*in Fig.* 14) *finds a possible square in its input* x, *word of* A^*, *in* $O(|x|\log|A|)$ *time in the worst case.*

Proof. All handlings on $\mathscr{C}(x)$, even if it is built inside the function square, take $O(|x|\log|A|)$ time in the worst case, i.e., when $\mathscr{C}(x)$ must be entirely built.

```
function square(a_1 ... a_n);
  j ← 0; q ← init; v ← 1; {empty word}
  while q ≠ last do
    i ← j; pos ← 0; r ← init;
    while q ≠ last and pos + r * a_{j+1} < i do
      j ← j+1; q ← q.a_j; r ← r.a_j; pos ← pos + r * a_j
    endwhile;
    if i = j then j ← j+1; q ← q.a_j; v ← a_j
    else if pos + (j − i) ≥ i then return('square')
    else u ← v; v ← a_{i+1} ... a_j;
      if left(u, v) or right(u, v) = 'square' then return('square')
      else if right(a_1 ... a_i, uv) = 'square' then return('square')
      endif
    endif
  endwhile;
  return('no square')
endfunction.
```

Fig. 14. Linear square searching algorithm.

The main point of the proof is the evaluation of the total cost of calls to left and right. At each pass in the loop the three calls, by Theorem 8.1, take a time proportional to $|u|+|v|$. Now it remains to note that, in the worst case, the sum of the lengths of the values of v is $|x|$ and is $<|x|$ for the values of u. □

8.4. Corollary. *On a given alphabet, square-freeness of a word x can be tested in time linear in the length of x.*

The factor transducer construction applied to square searching has given the surprising result in the above corollary. This result extends quite immediately to a linear detection of overlappings in words. We conjecture that the same result holds for other kinds of repetitions such as cubes and rational powers.

Appendix A. Product of square-free words

Figure A.1 contains an example of the function right whose definition appears in Section 8. The algorithm slightly improves the one given in [13]. Its correctness is based on two propositions. Their proofs as well as the time complexity evaluation are left to the reader. It may be shown that the maximum number of comparisons between two letters done by the function right on u and v is bounded by $3|v|$.

A.1. Proposition. *Let u, $v \in A^*$ be two square-free words such that* right$(u, v) =$ 'square'. *Let zyz be the shortest prefix of v such that y is a nonempty suffix of u. Then, z is the longest word which is both prefix and suffix of zyz. Furthermore, y is the longest common suffix of u and zy.*

```
function right(a_1 ... a_m, b_1 ... b_n);
  i ← n; test ← n+1;
  while i ≥ 1 do
    j ← i;
    while j ≥ 1 and m − i + j ≥ 1 and a_{m−i+j} = b_j do
        j ← j − 1
    endwhile;
    if j = 0 then return('square') endif;
    k ← i + j;
    if k < test then
      test ← k;
      while test > i and b_test = b_{j−k+test} do
          test ← test − 1
      endwhile;
      if test = i then return ('square') endif
    endif;
    i ← max{j, ⌈½i⌉} − 1
  endwhile;
  return('no square')
endfunction.
```

Fig. A.1. Function right.

A.2. Proposition. *Let u, v, y, z be as above. Let $v_1, v_2 \in A^+$ and $y', z' \in A^*$ be such that $|v_1| = |v_2|$, $v_1z'y'v_2z'$ is a prefix of v, zy is a proper prefix of $v_1z'y'$, y' is a suffix of u, and z' is the longest common suffix of v_1z' and $v_1z'y'v_2z'$. Then, the word zy is a proper prefix of v_1z' or $|zy| < \frac{1}{2}|v_1z'y'|$. Furthermore, zyz is a proper prefix of $v_1z'y'v_2$.*

References

[1] A.V. Aho and M.J. Corasick, Efficient string matching: An aid to bibliographic research, *Comm. ACM* **18** (6) (1975) 333–340.

[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[3] A.V. Aho and J.D. Ullman, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1979).

[4] A. Apostolico and F.P. Preparata, Optimal off-line detection of repetitions in a string, *Theoret. Comput. Sci.* **22** (1983) 297–315.

[5] J. Berstel, *Transductions and Context-Free Languages* (Teubner, Stuttgart, 1979).

[6] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen and J. Seiferas, The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.* **40**(1) (1985) 31–56.

[7] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler and R. McConnell, Finite automata for the set of all subwords of a word: An outline of results, *Bull. Europ. Assoc. Theoret. Comput. Sci.* **21** (1983) 12–20.

[8] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Comm. ACM* **20** (10) (1977) 762–772.

[9] M. Crochemore, Recherche linéaire d'un carré dans un mot, *C.R. Acad. Sci., Paris* **296** (série I) (1983) 781–784.

[10] M. Crochemore, Optimal factor transducers, in: *Proc. NATO Advanced Research Workshop on Combinatorial Algorithms on Words*, Maratea, Italy (1984) 31–43.

[11] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern-matching in strings, *SIAM J. Comput.* **6** (2) (1977) 323–350.

[12] M. Lothaire, *Combinatorics on Words* (Addison-Wesley, Reading, MA, 1983).
[13] M. Main and R. Lorentz, An $O(n \log n)$ algorithm for finding repetition in a string, Tech. Rept. CS-79-056, Washington State Univ., Pullman, 1979.
[14] M. Main and R. Lorentz, Linear time recognition of square-free strings, in: *Proc. NATO Advanced Research Workshop on Combinatorial Algorithms on Words*, Maratea, Italy (1984) 271–278.
[15] E.M. McCreight, A space-economical suffix-tree construction algorithm, *J. ACM* **23** (2) (1976) 262–272.
[16] J.H. Morris and V.R. Pratt, A linear pattern-matching algorithm, Tech. Rept. 40, Comput. Center, Univ. of California, Berkeley, 1970.
[17] M. Rodeh, A fast test for unique decipherability based on suffix-tree, *IEEE Trans. Inform. Theory* **IT-28** (4) (1982) 648–651.
[18] A.O. Slisenko, Determination in real time of all the periodicities in a word, *Soviet Math. Dokl.* **21** (2) (1980) 392–395.
[19] P. Weiner, Linear pattern-matching algorithms, in: *Proc. 14th Ann. Symp. on Switching and Automata Theory* (1973) 1–11.
[20] J. Ziv and A. Lempel, Compression of individual sequences via variable-rate coding, *IEEE Trans. Inform. Theory* **IT-24** (5) (1978) 530–536.